Universitetet i Oslo
Institutt for Informatikk

D. Karabeg, I. Yu

INF2220: algorithms and data structures

Mandatory assignment 1

:

Issued: 25. 08. 2014

Due: 18. 09. 2014

1 General description

In this assignment you will be implementing a *spell-check* application for (a subset of) the Norwegian language. If all words in the Norwegian language were to be present in all their forms, the dictionary would have to contain about one million words, so we select only a *subset* of these words (approximately 15.000 words). The words to be recognized by the spell-checker will be given to you in a *text file,* and the first part of this assignment will be to read the file and organize your dictionary.

The words that should be recognized by your spell-checker are located in the file `ordbok1.txt` on the inf2220 web-page. The number of words in this file will not surpass 15.000 and they come in random order (i.e., they are *not sorted* in advance, as you will see). The words are separated by newlines and blank spaces, and all the words are given in lowercase letters. Remember to convert user input to lowercase as well.

First we just want to do a simple look-up of the word the user has typed in, if we are unable to find the word in the dictionary, we will be looking for "similar words" to help him out. A more precise definition of "similar words" will be given.

NOTE: In this assignment everyone should follow the same programming pattern, and we will impose some structure that all must follow. Debates could be held about whether or not this is the optimal structure for a spell-checker but part of the reason for it being so is that we want to use central aspects of the curriculum. With a larger dictionary, other structures like B-trees would probably be a more likely candidate than the binary-trees we will be using in this assignment.

Data structure and access functions: "dictionary"

The words should be stored in a *binary tree.* Given that that the words are stored randomly in our input file, no steps have to be taken to balance the tree. As one consequence of that: the first word read remains the root of the tree).

After the binary tree has been built, a small change should be made to it; the word 'familie' (English: family) should be removed from the tree, then inserted again. This means that you have to implement a removal scheme, which does not destroy the property that a binary tree should have (it should be searchable).

When the binary tree is implemented, some statistics should be reported on how well their performance became. In particular the following should be reported:

Binary tree:

- The depth of the tree (length of the path to the node furthest away from the root)
- How many nodes are there for each depth of the tree.
- The average depth of all the nodes.
- Final and first word of the dictionary.

This can be computed while nodes are inserted into the tree, or by traversing the tree afterwards.

Using the dictionary

With the term *similar words* to X we mean:

- A word identical to X, except that two letters next to each other have been switched. (Example search term: laek; generated words: alek, leak, lake; where leak and lake are examples of English words that are generated)
- A word identical to X, except one letter has been replaced with another. (Example search term: laak; generated words: aaak, baak, ..., laaz; where leak is an example of an English word that is generated)
- A word identical to X, except one letter has been removed. (Example search term: lek; generated words: alek, laek, leak, ..., lekz; where leak and leek are examples of English words that are generated)
- A word identical to X, except one letter has been added in front, or at the end, or somewhere in between. (Example search term: leaak; generated words: eaak, laak, leak, leaa; where leak is an English word that is generated)

To check whether or not a word is contained in the dictionary, we use a *'lookup' operation.* When a user asks for a spell-check of a word X, the following procedure should be applied: First see if the word is located in the dictionary:

1. In this is case, give a positive answer and prompt the user for another word.

2. If not, generate similar words. If any of the similar words generated is found in the dictionary, print them out as suggestions to the user.

For those cases where the original word was not located in the dictionary (and multiple lookups for similar words have to be made), some statistics should be written out.

Statistics

- The number of lookups that gave a positive answer.
- Time used to generate and look for similar words.

User interaction

The focus in the assignment is not to design a particularly nice user interface. The program could just be an endless loop which prompts the user for a word, then tries to locate this word. If none can be found, suggestions base on the 'similar words' are printed. To abort this loop (quit the program) 'q' could be typed, which is not a valid Norwegian word anyway. A small menu should be printed out before the program goes into its endless loop of spell-checking. Exceptions should be handled in a reasonable fashion (for example, if a word consist of nothing but numbers, prompt the user for something which at least is made up of letters).

A few suggestions

When the program starts, something along these lines should be done:

- Read each word from 'ordbox.txt', which can be done easily using java.util.Scanner
- Insert the words into the binary tree
- Print a small menu ('q' to quit and so on...)
- Enter an endless loop which reads user input

Hint

Conversion between character arrays and strings is very simple in Java, to convert a string to a character array:

```
char[] alphabet = "abcdefghijklmnopqrstuvwxyzæøå".toCharArray();
```

and back again by initializing a new String object:

```
String str_alphabet = new String(alphabet);
```

This will come in handy during generation of similar words.

A good strategy is to implement functions that generate different types of similar words, then return these so you can investigate whether or not they are present in your dictionary. The combination of the two functions below, compute all words that have two letters next to each other switched (i.e., they will generate all similar words of type 1 from the definition).

```
public String[] similarOne(String word){

    char[] word_array = word.toCharArray();
    char[] tmp;

    String[] words = new String[word_array.length-1];

    for(int i = 0; i < word_array.length - 1; i++){
        tmp = word_array.clone();
        words[i] = swap(i, i+1, tmp);
    }

    return words;
}

public String swap(int a, int b, char[] word){
    char tmp = word[a];
    word[a] = word[b];
    word[b] = tmp;

    return new String(word);
}
```

Notice the use of the clone() function here, since arrays are passed by reference the original array is destroyed during the swap function, and we get a fresh copy of the original word by calling clone() on the original. It can also be a bit difficult to see in advance how long the String array should be with this approach, or how many similar words that will be created by such a swapping algorithm. In this case we are going to switch characters which are next to each other, and this can only be done in n-1 ways if the word has n characters (we actually count spaces between the characters, each of these give us two elements that can be swapped). If it is hard to compute the number of suggested words that will be generated in advance, more dynamic structures can be used, although that will naturally cost you time.

Handing in your assignment

The assignment should be carried out individually and you should submit your obligatory assignment using the Devilry system.

When you submit your assignment it should contain the following:

- Source code.
- Print-out of execution (on file utskrift.txt) which shows:
  - The different statistics
  - Spell-check of these words:
    - etterfølger
    - eterfølger

* etterfolger
* etterfølgern
* tterfølger

To get your assignment evaluated, it has to compile and run in a terminal-window on the *department's* Unix/Linux machines which means that it must have a text based interface. Your application can have a GUI as well, but it must also be possible to start a text based version of the program from the command line. The text based version should be the default version, and your GUI version should be started with a '-gui' option or so, if the user desires a graphical user interface. The source code should be documented, and self explanatory where comments are left out.

You should create a tar'ed or zipped archive where the top-level directory is your username.

So if your username is '`myusername`' and you want to tar it:

```
> mkdir myusername
> mv YourProgram.java utskrift.txt README.txt myusername/
> tar -czf myusername.tgz myusername/
```

This will give you a file called `myusername.tgz` which holds all the content of the assignment in a gzipped (compressed) tar-file (tape archive).

If you are writing the oblig on your own computer, and cannot produce TAR archives, you may also use ZIP archives.

The archive should be submitted in Devilry by the deadline. Information regarding the deadline and the assignment should show up when you log into Devilry and enter the Student page. If the course does not show up in your list of courses, you should contact the student administration.

Compiled classes (.class files) should not be included in your tar-archive, and if there are any options required (or CLASSPATH settings) in order to compile, these should be described in the file README.txt. If you have any questions about the format, please do not hesitate to ask your teaching assistant.

Good luck!